On the Optimal Order of Reading Source Code Changes for Review

Baum, Tobias ; Schneider, Kurt ; Bacchelli, Alberto

Abstract: Change-based code review, e.g., in the form of pull requests, is the dominant style of code review in practice. An important option to improve review's efficiency is cognitive support for the reviewer. Nevertheless, review tools present the change parts under review sorted in alphabetical order of file path, thus leaving the effort of understanding the construction, connections, and logic of the changes on the reviewer. This leads to the question: How should a code review tool order the parts of a code change to best support the reviewer? We answer this question with a middle-range theory, which we generated inductively in a mixed methods study, based on interviews, an online survey, and existing findings from related areas. Our results indicate that an optimal order is mainly an optimal grouping of the change parts by relatedness. We present our findings as a collection of principles and formalize them as a partial order relation among review orders.

DOI: https://doi.org/10.1109/ICSME.2017.28

On the Optimal Order of Reading Source Code Changes for Review

Tobias Baum and Kurt Schneider
FG Software Engineering
Leibniz Universität Hannover
Hannover, Germany
Email: [firstname.lastname]@inf.uni-hannover.de

Alberto Bacchelli
ZEST – Zurich Empirical Software engineering Team
University of Zurich
Zürich, Switzerland
Email: bacchelli@ifi.uzh.ch

***Abstract*—Change-based code review, e.g., in the form of pull requests, is the dominant style of code review in practice. An important option to improve review's efficiency is cognitive support for the reviewer. Nevertheless, review tools present the change parts under review sorted in alphabetical order of file path, thus leaving the effort of understanding the construction, connections, and logic of the changes on the reviewer. This leads to the question: How should a code review tool order the parts of a code change to best support the reviewer? We answer this question with a middle-range theory, which we generated inductively in a mixed methods study, based on interviews, an online survey, and existing findings from related areas. Our results indicate that an optimal order is mainly an optimal grouping of the change parts by relatedness. We present our findings as a collection of principles and formalize them as a partial order relation among review orders.**

I. Introduction

Code Review, particularly in the form of Inspection [1], has been shown to be an effective software quality assurance technique for decades. In recent years, its use in industry is converging towards 'change-based code review' [2],[1] a variant of code review embedded into the software development process such that the changes performed in some development task define the review scope. Change-based code review has gained a vast popularity [5], especially in the form of pull-based software development [6] as provided by GitHub [7], therefore ways to improve the effectiveness and efficiency of change-based code review can lead to a significant impact.

As change-based code review is usually supported by computerized tools, better computerized cognitive support for the reviewer is a promising avenue to gain such improvements [8]. One possibility for better cognitive support lies in presenting the code changes to be reviewed in a better way: Current review tools usually list the change parts alphabetically ordered by file path. Barnett et al. [9] and Baum and Schneider [8] indicate that this order could be sub-optimal in many cases, especially for larger changes; we present further evidence for this claim in Section III. The natural follow-up question is: *How should the parts of a code change be ordered to best support the reviewer during change-based code review?*

In this paper, we present a work aimed at answering this question. We follow a theory-generating methodology [10] and combine input from multiple data sources: (1) Log data from 292 tool-based review sessions in industry; (2) task-guided interviews with 12 professional developers; (3) related work from fields such as cognitive science and program comprehension; and (4) a task-based survey answered by 201 reviewers. We present our findings as a collection of 'principles' given in natural language and we formalize them into a theory[2] to improve their verifiability and utility for follow-up studies.

With our work we contribute the following:

- Further empirical support that improving the order of presenting code changes for review is worthwhile
- Empirically grounded principles that describe which order of presenting code changes helps to achieve better review efficiency
- A theory derived by formalizing these principles

II. Methodology

A. Research Questions

Although we used an iterative methodology, we present our results structured linearly along 4 research questions.

Reviewers can navigate the code in two ways: On their own, driven by hypotheses they form along the way, or guided by the order presented by the tool. There is qualitative evidence that both occur in practice and that the current tool order is sub-optimal [8], [9]. To add further qualitative as well as quantitative support to this claim, we set to answer:

> **RQ1.** How relevant to reviewers is the order of code changes offered by the code review tool? (Section III)

Given that the current order is perceived as sub-optimal, we then focus on empirically defining what makes a better order:

> **RQ2.** Are there principles for an optimal order of reading code changes under review? (Section IV)

Those principles, if any, would provide a human-readable impression of what is meant when talking about presenting the

[1] also 'modern code review' [3] or 'continuous differential code review' [4].

[2] Writing about "generating theory" in this paper, we use the word theory in the sense of Sjøberg et al. [10]: An empirically-based description of constructs and propositions that can be used to derive testable predictions. As a "middle-range theory" it involves abstraction but is still closely linked to observations.

changes in a better order. To be implemented in software or used for predictions, they need to be specified more precisely:

RQ3. How can the notion of 'better order for code review' be formalized as a theory? (Section V)

Finally, even though no previous study on the optimal order of reading changes for change-based code review has been conducted, our investigation shares similarities with other studies (e.g., by looking at the influence of structure on understanding or by proposing techniques to make reading code in code reviews more efficient). We consider these studies not only for informing our theory, but also to triangulate it:

RQ4. How does the theory fit within the related work and existing evidence? (Section VI)

B. Research Method

Our approach was inspired by methods to iteratively generate theory from data, most notably Grounded Theory [11], [12].[3] We started with interviews as a flexible means to gather rich data and triangulated our findings with empirical results from related fields as well as with an online survey. Furthermore, we collected log data from code reviews in industry to support the claim that better automatic ordering of changes can help to improve review. We detail the data sources and their connection to the answers, as in Figure 1.

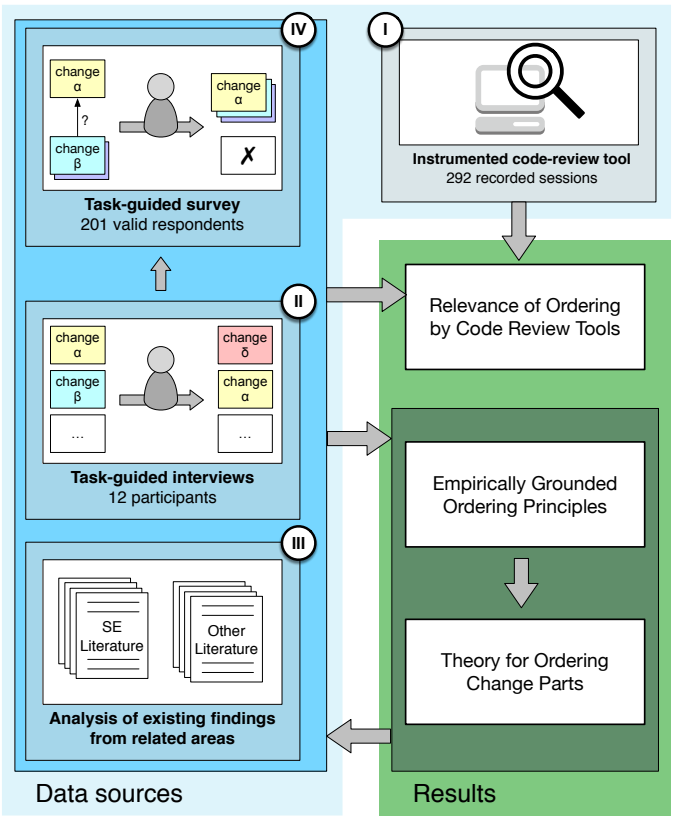

Fig. 1. High-level view of the research method

Logged review navigation (Point I in Figure 1): Past studies provided qualitative evidence that the current standard order of review tools, i.e., alphabetical by path, is not optimal [9], [8]. To triangulate these findings and to gain more quantitative information, we analyzed data from a medium-sized software company that develops a software product with about 20 developers. To gain insights into the navigation patterns, we instrumented a code review tool to log detailed interaction data during reviews (software telemetry [13]). The data used in the current study consists of 292 reviews collected during fall 2016. To determine the size of code changes, we analyzed the company's versioning system. The data set is available [14].

Task-guided Interviews (Point II): The main method of data collection in the early phases of the study was a special form of *task-guided* interview. We prepared exemplary code changes from real world projects and printed each part of the changes in form of a two-pane diff on a different piece of paper. Interviewees were asked to sort the shuffled parts for a change into the order believed to be best for review. A short printed description of the participant's task and the purpose of the change was also handed out. While sorting, the participant was asked to think aloud. When the participant had finished, the interviewer explicitly asked for a rationale for the given order. After that, the interviewer presented an alternative order (either taken from an earlier interview or prepared by the researchers before the session) and the participant was asked to explain why his/her order was better than this alternative. In most interviews, we repeated this comparison with yet another order, in other interviews we repeated the whole procedure with a different code change.

We used three different change sets (A, B, and C) sampled from real-world software systems. Our main criteria for selecting the changes were their size/complexity (manageable, yet not trivial) and their content (based on our preliminary hypotheses). Changes consist of 7 to 10 change parts, mainly in Java files, but also in XML and XML Schema files. In the interviews, the participants were familiar with the codebase for changes B and C, but not for A. Details on the changes are in the study's material [14].

All of the sampled interview participants had good programming knowledge, but not all of them were experienced reviewers. Table I presents the participant's demographics. Seven of the interviews were performed by the first author of this paper and five by the last. In the first four interviews, the researcher took interview notes; all of the other interviews were recorded and later transcribed.

The interviews resulted in two types of data: the orders declared optimal by the participants and the interview transcripts. The interview transcripts were analyzed by open coding augmented by memoing. The codes were then refined and checked in a peer card sort session [15], [16] performed jointly by the

[3]We do not regard our study as a Grounded Theory study in the narrow sense but rather as a mixed methods study, among other things because our research question was largely fixed before starting the data collection.

TABLE I
INTERVIEW PARTICIPANTS: ID, EXPERIENCE, AND CHANGE SET

ID	Dev. exp. (in years)	Change set	ID	Dev. exp. (in years)	Change set
I1	2	A	I7	23	B+C
I2	8	A	I8	3	A
I3	3	A	I9	10	A
I4	3	A	I10	5	A
I5	7	A	I11	10	A
I6	5	B+C	I12	10	A

first and third author (Point 5), followed by further selective coding. The sequences given by the participants were included in the card sorting and also analyzed programmatically to systematically search for nonrandom patterns. We furthermore used them later to check the formalization of the theory. During the whole research process, we used memoing to capture ideas and preliminary hypotheses. The study material (i.e., the interview guide and transcripts, the change parts, and an Atlas.TI project with the codes and memos) is available [14].

Existing Findings in Related Areas (Point III): One of the guidelines in Grounded Theory is that "other works simply become part of the data and memos to be further compared to the emerging theory" [12]. With this mindset, we used existing findings from related areas to inform and triangulate our theory. The selection of related fields was guided by theoretical sampling, e.g., by looking for works on hypertext after the importance of relations began to emerge. When sampling from a research field, we tried to get a good overview, but we did not perform systematic literature reviews in the narrow sense.

Task-based Survey (Point IV): After formulating preliminary hypotheses, we conducted an online survey. It contained *task-guided* confirmatory questions to challenge the preliminary hypotheses and exploratory questions to develop the theory further. We defined the target population as 'software developers with experience in change-based code review' and included a set of questions to filter respondents accordingly.

We used established guidelines for survey research [17] to formulate the questions and structuring the survey. The main part consisted of tasks asking respondents to declare their opinion regarding different code orders for review. In addition, besides the filter questions, the survey contained 3 questions on the participant's navigation behavior during code review (for RQ1) and 4 questions to analyze potential confounding factors, such as the used review tool and programming language. All the questions were optional, except for the filter questions.

The main, task-guided part consisted of four pages; Figure 2 shows an extract of one. Every page started with the abstract description of a code change (Point 1 in Figure 2) and ended with a free text question for further remarks (Point 2). Between that, a selection of three types of questions was included, which asked for: the participant's preferred order of the change parts (Point 3), a comparison of two orders pre-selected based on the research hypotheses (Point 4), and an assessment of the usefulness of a set of orders on a 4-point Likert scale (not shown in Figure 2). The order in which the change parts and

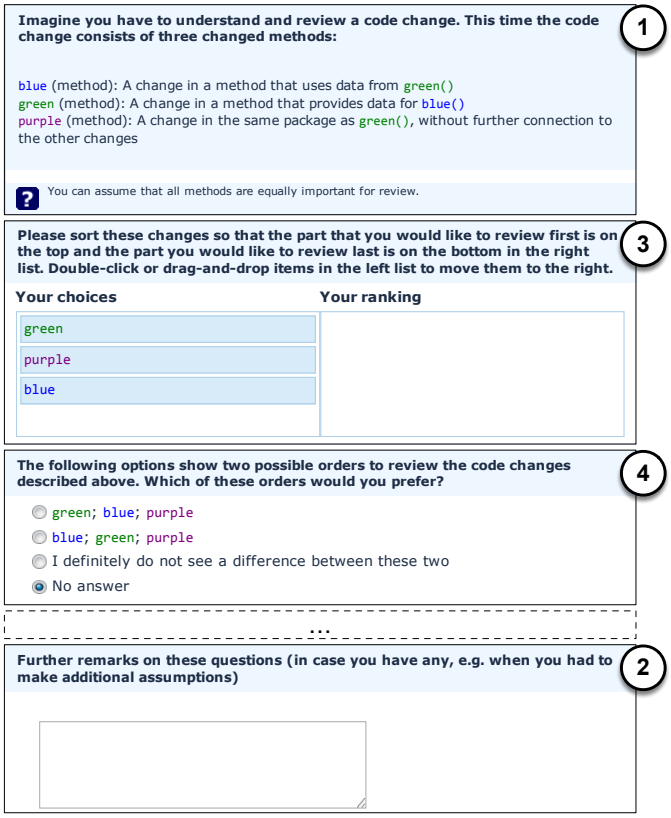

Fig. 2. Example main page from the survey (data-flow variant/Situation 2a)

proposed orders were presented was randomized.

We used eight pre-tests with software developers to iteratively optimize the survey. The creation and testing of the survey took seven weeks; the final survey ran for five weeks.

To invite software developers to our survey, we randomly sampled active GitHub users and invited developers from our professional networks. We invited a total of 3,020 developers. The initial filter questions were answered by 238 people (response rate: 8%), of which 201 were part of the target population. Not all participants completed the survey or answered all questions. We excluded participants if values for the respective hypothesis/research question were missing or if an answer was inconsistent with one of their earlier answers. Therefore, the total number of answers differs for the analyses.

Of the respondents, 97% program and 85% review at least weekly, so we did not take further measures to account for varying practice of the participants. 57% of the respondents have only one or two years of experience with regular code reviews; we included these participants unless otherwise noted, but only after statistically showing their answers to be distributed like those of the respondents with 3 years or more of experience. The influence of sampling through GitHub is clearly visible: A majority (102 of 177) uses GitHub pull requests for reviewing and JavaScript (66 of 133) or Java (41 of 133) as a programming language. 89% (117 of 132) develop software commercially and 72% (95 of 132) participate in open source. The survey and the data sets are available [14].

C. Limitations

We describe limitations of our study, for both the whole study and the different methods of data collection.

If our study was a *theory-testing* study, it would be severely limited by two assumptions underlying our argumentation: (1) For the interviews and survey: What *is* a good tour and what experienced developers *think* is a good tour coincides to a large degree. (2) For our usage of related work: Findings from natural language reading and program comprehension can be transferred to code change comprehension. As our goal is efficient *generation* of theory instead, we deem these assumptions to be acceptable. One of the most important tasks in future work is to make the reliance on these assumptions unnecessary by directly testing the theory.

The main limitation of the collected log and repository data is that it comes only from a single company and code review tool. Since the general tendency we found in this data was also supported in the survey, we deem this as a limitation to the generalizability of the exact numbers only.

One of the greatest risks in the interviews and the survey was to accidentally introduce a bias for a certain order. We took several measures to counter this risk: In the interviews, we shuffled all change parts and used random words instead of numbers as IDs for the distinct parts. We did not remove line numbers as they could be part of a sensible ordering strategy. For the survey, we used randomized orders in the questions and in the descriptions, and we used colors as part IDs. Color names instead of random words were found to be easier to understand in the pre-tests.

To avoid the anchoring effect [18] in both the interviews and surveys, we first asked for the participant's preferred order before presenting other orders. By describing the steps of the interview in an interview guide and by videotaping some sessions, we increased intersubjectivity. Mitigating researcher bias was one of the reasons to perform a joint card sort.

During survey creation, we checked against published guidelines [17] and performed several rounds of pre-testing. Nevertheless two factors probably introduced some noise into the data: (1) Understanding the abstract situations was still a problem for some participants (some respondents indicated that they left the respective questions empty, but others might have answered without having understood the described situation); (2) the drag and drop support in the ranking widget (used for the questions of type Point 3 in Figure 2) had to be used with a certain care to avoid unintended results.

A negative side-effect of our sampling method is that the sample should be regarded as self-selected; we included a number of questions into the survey to characterize the sample and check for influencing factors and analyzed the answers.

The generalizability of our results to the population of users of change-based review is probably quite high, mainly due to the large number of participants in the survey. The sample of distinct code changes we used is much smaller, which could impede generalizability in this regard. Specifically, the set of relations given in Section IV may be incomplete.

III. The Relevance of the Order by the Tool

Our first research question seeks to understand the relevance of the ordering of changes proposed by the code review tools. To answer this question, we gathered opinions in interviews, log data from tool-based reviews in industry, and estimates from our survey's respondents.

The log data contains traces of files visited in a review session. By comparing these files to the alphabetical order, we identify whether the user followed the (alphabetical) order of the review tool and types of deviations. In 156 of 292 studied review sessions (53%), the user started with the file presented first by the review tool. When reviewing further, 3,071 of 8,254 between-file navigations (37%) took the reviewer to the next file in the tool order; moreover, in 162 (55%) of the review sessions, the reviewer visited additional files that were not part of the change set. We interpret that the hyperlinking and search features of the IDE help the reviewer navigating, after a place to start is found.

For small changes, the presentation order may have a negligible effect, as the number of permutations and the cognitive load for the reviewer grows with change size. In a study at Microsoft, Barnett et al. [9] found a median size of 5 files and 24 diff regions in changes submitted for review. The review sessions we analyzed are even larger, with a median size of 11.5 files per task. About 54% of the reviews had a scope of 10 or more files. We could not find a statistically significant pattern connecting review size and navigation behavior.

In our interviews, we asked the participants' opinion on an alphabetical order for review. They were either neutral or negative about it, e.g., "*Well, I don't think file based is a good order* [...] *or alphabetical order is definitely not a good order.*"$_{I9}$ "*I mean it's clear that GitHub doesn't have any intelligence behind the way that it presents you the reviews, currently, so even a small improvement is welcome.*"$_{I10}$ And although we did not ask the participants of our survey for it, some left a remark at the end of the survey, for example: "*I've never thought about ordering of changes in code review tools. But while filling this survey I started thinking that proper ordering could make reviewing of code much simpler.*"$_{S270}$

When a code review is performed jointly with the author of the code, the author can guide the reviewer through the code. Therefore, we asked about joint reviews in our survey. Of 167 participants answering this question, 32 (19%) perform reviews 'often or always' together with the code's author. We asked the remaining participants about their behavior in two situations: (1) When starting the review and (2) when in the midst of the review. 132 respondents answered these questions and a large fraction reported to use the tool's order 'often or always': 97 (73%) for the start and 87 (66%) for the middle of the review; Figure 3 reports the details. We found that there is a tendency for more experienced reviewers and for reviewers with IDE-integrated review tools to use the tool's order less often,[4] but the general picture stays the same for all

[4]Always continuing in tool order: experienced: 6/62, inexperienced: 18/70; IDE-based: 1/24, web-based: 23/106

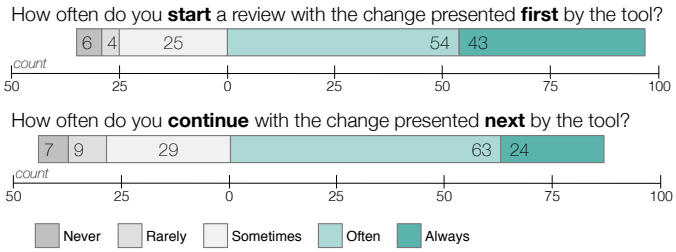

Fig. 3. Survey results: Relevance of the ordering offered by the tool

sub-populations we studied.

Summing up, in a significant number of change-based code reviews, the reviewers use the order in which the code changes are presented by the review tool to step through the code, although they report to regard this order as sub-optimal for efficient understanding and checking of the code. The size of a code change under review is often small, but in a notable number of cases not small enough to make the effect of the order irrelevant. The problem is more pronounced for less knowledgeable reviewers. A code review tool should, therefore, present the changes in an order that is well-suited for a human reviewer. The results of our next research questions describe how such an order should look like.

IV. Principles for an Optimal Ordering

Our second research question seeks to extract general principles to guide the ordering of changes to support code review.

A. General Principles

The interview participants believe that certain orders are better suited for code review than others. But the choice of the optimum is subjective. However, participants generally acknowledged that other orders are good, too, and believe that a number of orders will be similar in terms of review effectiveness and efficiency: "*I don't necessarily think this is worse. It's more a different point of view.*"$_{I4}$ "[This order] *probably makes sense if you're super-deep into the system.*"$_{I10}$

Following the 'tour/path' metaphor used in other publications [19], [20], we use the term 'tour' in the following to denote a permutation of the change parts under review. Our 'change parts' and change hunks from the version control system are related, but do not have to be identical, as will be further detailed in Section V.

Based on the combination of the different data sources, it emerges that an order that obeys the principles described in the following is perceived to lead to better review effectiveness and efficiency compared to other orders.

> *Principle 1:* Group related change parts as closely as possible.

By grouping related change parts together, we avoid context switches and reduce the cognitive load for the reviewer. Additionally, we ease the task of comparing neighboring change parts to spot inconsistencies or duplications: "*So here, seems a bit like a code clone.* [...] *And this is actually why I think it is really cool to have these two* [related change parts] *together.*"$_{I10}$ "*If I was to return on this one, I would have to switch the context, which is bad.*"$_{I8}$ "*I think a review tool should try to group changes that 'logically' belong together.*"$_{S44}$ "*Unrelated things should not get in the way of related things.*"$_{S296}$ As shown in Table II, Principle 1 is also well supported in the survey results.

TABLE II
Survey results: Confirmatory questions for Principle 1.

Preference	Sit. 2a + 2b[1]	Sit. 2a (Only data-flow)	Sit. 2b (Only attr. decl.-use)	Sit. 4[2]
close	102	67	35	102
not close	14	4	10	6
no preference	7	4	3	
Total resp.	123	75	48	108

[1] Only one of Situation 2a (data-flow relation) and Situation 2b (declaration-use relation) was shown selected by chance.
[2] Results for Situation 4 deduced from the user-given order (Point 3 in Figure 2), therefore "no preference" does not occur.

TABLE III
Survey results: Comparison of different ways to order change parts related by call-flow (one callee, four callers), used for Principle 2 and 6.

Order strategy	Count among best rated	Count among worst rated	Mode (prevalent answer)
bottom-up	111	16	very useful (100 times)
top-down breadth-first	35	61	somewhat useful (49 times)
top-down depth-first	34	67	somewhat useful (56 times)
no sensible rule	10	112	not very useful (55 times)[1]
Total resp.	130	130	130

[1] For experienced reviewers, the mode is "not at all useful" (21 of 45).

> *Principle 2:* Provide information before it is needed.

This allows the reviewer to better understand the change parts: "*Without the knowledge if this attribute is required or optional, I can't tell if the mapper is correctly implemented.*"$_{I7}$ "*Blue depends on green, so it's useful to know what green is before reviewing blue.*"$_{S248}$

Although we did not include a confirmatory question for Principle 2 in the survey, we interpret one of the results as attributable to it: The respondents showed a clear tendency towards going bottom-up along the call-flow relation, with 111 of 130 (85%) rating bottom-up as preferred (see Table III).

> *Principle 3:* In case of conflicts between Principles 1 and 2, prefer Principle 1 (grouping).

When reviewers come across a change part where they need knowledge they do not yet have, they need to make assumptions (at least implicitly). As long as the related information-providing change parts are coming shortly

TABLE IV
SURVEY RESULTS: CONFIRMATORY QUESTIONS FOR PRINCIPLE 3.

Preference	Sit. 2a + 2b[1]	Sit. 2a (Only data-flow)	Sit. 2b (Only attr. decl.-use)
prefer closeness	82	57	25
prefer direction	26	12	14
no preference	11	5	6
Total resp.	119	74	45

[1] Only one of Situation 2a (data-flow relation) and Situation 2b (declaration-use relation) was shown selected by chance.

afterward, they can then check the assumptions against reality: "*The order doesn't actually influence me that much.*"$_{I11}$ "*Maybe the order was not what I preferred, but the groupings of the snippets made sense.*"$_{I5}$ "*The only thing that matters here is that purple and gold appear one after the other, whichever first.*"$_{S392}$ Principle 3 is supported in the survey, although less than Principle 1: 76% of the respondents who indicated a preference preferred closeness (see Table IV).

> *Principle 4:* Closely related change parts form chunks treated as elementary for further grouping and ordering.

Principle 4 was needed to explain some of the interview results and is supported in the literature on cognitive processes, but it did not emerge explicitly from the interviewees' statements. Therefore, we included two exploratory questions in the survey to investigate it (Situation 3). The tour using chunking to let the relations point in the preferred direction was chosen as better by 35 of 50 respondents (70%) in one and 38 (76%) in the other question.

> *Principle 5:* The closest distance between two change parts is "*visible on the screen at the same time.*"

Seeing two closely related change parts directly after another is good, but seeing them both at the same time is better: With the latter, the cognitive load is minimal and inconsistencies can be spotted. As our participants put it: "*The most useful presentation would be to display all* [5 related] *changes at once and to allow the user to navigate freely*"$_{S44}$ "*I'd prefer to have them both* [...] *on screen, ideally.*"$_{S54}$ "*I'd make* [the description] *stay on top, wherever I look at.*"$_{I8}$

> *Principle 6:* To satisfy the other principles, use rules that the reviewer can understand. Support this by making the grouping explicit to the reviewer.

An order that the reviewer does not understand can "*break his line of thought*" and lead to disorientation. Making the grouping explicit helps the reviewers to understand it, to form expectations and to divide it into parts they can handle separately. "*We're going back from something that is more specific to something that is generic. And that kind of breaks my line of thought.*"$_{I10}$ "[This order] *doesn't have a specific pattern, at least none that I can immediately identify.* [...] *This is bad*"$_{I5}$ "*If the parts had been grouped, the groups made visible and ideally given sensible names, I would have been able to understand the ordering better.*"$_{I5}$

When asking the survey participants to rate bottom-up vs top-down tours, we also included a tour that was not based on a "sensible" rule. This option was rated among the worst by 112 of 130 respondents (86%), thus supporting Principle 6.

The common guideline to 'keep commits self-contained' is a special case of Principle 1 combined with Principle 6. In this case, a commit is an explicit group of related change parts.

The importance of a change part for code review varies, e.g., some change parts are more defect-prone than others. The participants took this importance into account to varying degrees. Some used it as an important ordering criterion, while others did not. A lot of the variation in participant's orders from the interviews is due to these differences in the assessment and handling of unimportance.

B. The Macro Structure: How to Start and How to End

At the very beginning of the review, the reviewer should learn about the requirements that led to the change. Many also wanted to get some kind of overview at the start ("First introduction to understand the context, then the crucial part"$_{I2}$). An example of usage, e.g., a test case, can help to achieve this.

We could observe several tactics (T) among our interviewees on how to proceed after that, i.e., start with: (T1) something easy, (T2) a natural entry point, e.g. GUI, Servlet or CLI, (T3) the most important change parts, (T4) new things, (T5) change parts that "don't fit in", if any. Of these tactics, T1 and T2 often suit Principle 2 better, i.e., to provide information before it is needed. In contrast, T3, T4 and T5 are heuristics to visit more important/defect-prone change parts early.

Some participants gave tactics for the end of the review, too: (1) End with a wrap-up/overview (e.g. a test case or some other example of usage putting it all together), or (2) put the unimportant rest at the end.

C. The Micro Structure: Relations between Change Parts

Principle 1 states that related change parts should be close together. The participants gave a number of different types of "relatedness", e.g.: (1) Data flow, (2) call flow, (3) class hierarchy, (4) declare & use, (5) file order, (6) similarity, (7) logical dependencies, and (8) development flow. A more detailed description of the relation types can be found in the supplemental material [14].

Most of these relations are inherently directed (e.g. class hierarchy or data flow), while others are undirected (e.g. similarity). For many of the directed relations, we observed a preferred direction (e.g. to put the declaration of an attribute before its use); for others—mainly for call flow—the preferred direction seems to be more subjective. Many interview participants preferred to go top-down from caller to callee, but others also talked about going bottom-up from callee to caller. In contrast, the survey results support bottom-up (see

TABLE V
DEFINITIONS OF THE CONSTRUCTS

Construct	Description
Code change	The "code change" consists of all changes to source files performed in the "unit of work" [2] under review. This also includes auxiliary sources like test code, configuration files, etc. The code change defines the scope of the review, i.e., the parts of the code base that shall be reviewed. With task or user story level reviews, a code change can consist of multiple "commits".
Review efficiency	Review efficiency is the number of defects found per review hour invested (definition adapted from [21]).
Review effectiveness	Review effectiveness is the ratio of defects found to all defects in the code change (definition adapted from [21]).
Defect	In the context of this study, a defect is any kind of true positive issue that can be remarked in a review. This encompasses faults as defined in the IEEE Systems and Software Engineering Vocabulary [22], but also for example maintenance issues. This definition is sufficient as we are primarily interested in relative differences in the number of defects.
Change part	The elements of a code change are called "change parts". In its simplest form, a change part corresponds directly to a change hunk as given by the Unix diff tool or the version control system. When some part of the source code was changed several times in a code change, a change part can span more than two versions of a file. It could be beneficial to split large change hunks into several change parts, e.g., when the hunk spans several methods.
Tour	A tour is a sequence (permutation) of all change parts of a code change.
Relation (between change parts)	There can be "relations" between change parts. A relation consists of a type (e.g. call flow, inheritance, similarity; see Section IV-C) and an ID that allows distinguishing several relations of the same type (e.g. the name of the called method). There are relations of differing strength, but we do not take this into account in the current formal model. Change parts (as vertices) and relations (as edges) define a graph with labeled edges, the "part graph". There are directed as well as undirected relations. We model undirected relations as two directed edges so that the graph is directed. There can be multiple edges between two change parts, but their labels have to be distinct. We further demand that the graph has no loops. A mechanism similar to the one used by Barnett et al. [9] can probably be used to get from the syntactic level to the relation graph.
Grouping pattern	The grouping and ordering preferences of a reviewer are modeled as "grouping patterns". A grouping pattern combines a matching rule that identifies a subset of change parts in the part graph and a function $rate$ to provide a rating for a permutation of the matched change parts. We found only one family of grouping patterns to be sufficient to describe our data so far: A "star pattern" (see Figure 4) matches a core vertex and all vertices (at least one) that are connected by an edge with a given relation type and the same ID to the core. In the "bottom-up" case, the rating function assigns a high rating (e.g. 1) to all sequences that start with the core and a low rating (e.g. 0) to all others.

Table III). We interpret that a simple global rule of "always prefer bottom-up/top-down" probably does not exist.

Another distinction between the relations is whether they are binary or gradual. For a binary relation, like call flow, there are only two possibilities: Either there is a relation or there is none. For a gradual relation, like similarity, the distinction between related and unrelated is fuzzier.

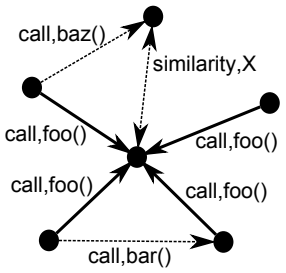

Fig. 4. Example of a Star Pattern (Thick Edges) in a Change Part Graph

V. A THEORY FOR ORDERING CHANGES FOR REVIEW

The principles and findings described in the previous sections detail what makes a good order of changes for code review, but to implement them in software or to test the hypotheses, they are still too vague. Therefore, we formalize them based on the guidelines for building theories in software engineering by Sjøberg et al. [10], i.e., by giving our theory's scope, constructs, propositions, and the underlying explanations.

A. Scope and Constructs

The scope of our theory is change-based code review [2]. The theory has been developed based on code written in object-oriented languages; it possibly has to be adapted to be applicable to other programming paradigms.

The constructs of our theory are detailed in Table V.

B. Propositions

The goal of this section is to define a partial order $\geq_T \subseteq Tour \times Tour$ (in words: is better than) between tours that captures our notion of the utility of an order for review. For every pair of tours t_1 and t_2 it holds (other things being equal):

$$\forall t_1, t_2 : t_1 \geq_T t_2 \Rightarrow \\ (\text{reviewEfficiency}(t_1) \geq \text{reviewEfficiency}(t_2) \wedge \\ \text{reviewEffectiveness}(t_1) \geq \text{reviewEffectiveness}(t_2))$$

The proposition above states that a tour that is better than another in terms of $\geq_T$ will not be worse in terms of review efficiency or effectiveness. We also expect a stronger proposition to hold, namely that there are tours where a better ranking in terms of $\geq_T$ means better review efficiency:

$$\exists t_1, t_2 : t_1 \geq_T t_2 \wedge \neg (t_2 \geq_T t_1) \Rightarrow \\ \text{reviewEfficiency}(t_1) > \text{reviewEfficiency}(t_2)$$

The definition of $\geq_T$ is parametric; based on a set P of grouping patterns. Different preferences of reviewers can be captured by changing this set P. It also depends on the part graph g, which we assume to be implicitly known. By using a

partial order, we allow for two tours to be incomparable, which we exploit when there is yet no sufficient empirical evidence to base the comparison upon.

The relation $\geq_T$ is defined based on a helper construct, the 'match set' (MS) of a tour. The MS consists of all occurrences of a grouping pattern in a tour. A grouping pattern match occurs in a tour when *all* vertices matched for the pattern in the part graph are direct neighbors in the tour. Structurally, a pattern match is a pair (p, v) of a grouping pattern p and the set of matched change parts v.

A tour is better than another when its MS is better, i.e., when it has more matches or the same matches with higher ratings, as given by the grouping pattern's rating function (rate):

$$\begin{aligned} t_1 \geq_T t_2 &\iff \mathrm{mS}(t_1, g) \geq_{MS} \mathrm{mS}(t_2, g) \\ &\iff \mathrm{mS}(t_1, g) \supset \mathrm{mS}(t_2, g) \vee \\ &\quad (\mathrm{mS}(t_1, g) = \mathrm{mS}(t_2, g) \wedge \\ &\quad \forall m \in \mathrm{mS}(t_1, g) : \mathrm{rate}(m, t_1) \geq \mathrm{rate}(m, t_2)) \end{aligned}$$

The inclusion of all pattern matches from the sequence of change parts in a tour into the match set mainly formalizes Principles 1 (group related parts) and 3 (prefer grouping). To also formalize Principle 4 (chunking), we introduce the notion of shrinking a tour (and the corresponding part graph) by combining change parts: The function $shrink : Tour \times PartGraph \times \wp(ChangePart) \rightarrow Tour \times PartGraph$ creates a new tour by removing all change parts contained in the set given as the third parameter and replaces them with a composite part. On the part graph, it also combines all given change parts into the composite part. Edges that pointed to one of the removed parts now point to the composite part. If this leads to duplicate edges or loops, they are combined/removed.

We conclude defining the recursive function $mS : Tour \times PartGraph \rightarrow MS$ (pM stands for $patternMatches$, i.e., the matches for a pattern in a tour given a graph):

$$\mathrm{mS}(t, g) := \bigcup_{p \in P} \left(\mathrm{pM}(p, t, g) \cup \bigcup_{m \in \mathrm{pM}(p,t,g)} \mathrm{mS}(\mathrm{shrink}(t, g, m.v)) \right)$$

Table VI shows how the formalization reflects the principles and other empirical findings presented in Section IV.

C. Explanation

To end the presentation of the theory, we will now summarize and extend its rationale: It is based on the assumption that the efficiency and effectiveness of code review (with a fixed number of reviewers) is largely determined by the cognitive processes of the reviewers. The reviewer and the review tool can be regarded as a joint cognitive system [23], and the efficiency of this system can be improved by offloading cognitive processes from the reviewer to the tool. The relevant cognitive processes can be divided into two parts: Understanding the code change, and checking for defects. The way in which the changes are presented to the reviewer influences both. A good order helps understanding by reducing the reviewer's cognitive load and by an improved alignment with human cognitive processes (hierarchical chunking and relating). It helps checking for defects by avoiding speculative assumptions and by easing the spotting of inconsistencies.

VI. The Theory in the Context of Related Work

We contextualize our theory within the related work in software engineering and comprehension research.

A. Reading Comprehension for Natural Language Texts

Brain regions responsible for language processing are also active during code comprehension [24]. There are differences in the activation patterns between code and text, but these become less pronounced with programming experience [25]. Therefore, we used several studies from the large body of research on reading comprehension and the understandability of natural language texts to inform our theory.

The "Karlsruhe comprehensibility concept" [26], an extension of the "Hamburg comprehensibility concept" [27], summarizes multiple studies on factors influencing the comprehensibility of natural language texts. It names six influencing factors/dimensions: Structure, concision, simplicity, motivation, correctness and perceptibility. Our approach to improving code ordering mainly targets the "structure" dimension.

The positive impact of a sensible, explicitly recognizable or presented text structure is also reported in other studies (e.g. [28], [29] and [30]). A good text structure is "coherent", i.e., parts of the texts hang together in a meaningful and organized manner [30]. Reading scrambled paragraphs takes more time, and is detrimental to recall quality when there is a time limit [31]. Presenting news and corresponding explanations in a clustered way can improve the understanding and interest of a reader [32]. And there is evidence that stories are mentally organized in a hierarchical fashion [33].

The aforementioned results fit to ours, but there is also some evidence to the contrary: McNamara et al. found that a less coherent structure can improve the learning of knowledgeable readers from a text, presumably because they have to think more actively [34]. The same could be true in our case, with a sub-optimal structure forcing the reviewer into a more active role. This underlines the need to empirically test our predictions in future work.

B. Hypertext: Comprehension and Tours

Our theory is based on the assumption that the relations between change parts are an important factor in determining the optimal tour. The resulting part graph shares many similarities to a hypertext. Hypertext research has studied how different link structures and different presentations of the structure influence a reader's interest and understanding (e.g. [35]). We cannot influence the link structure in our case, but we can influence the presentation, and a hierarchical presentation has been found to be beneficial [36]. It has also been found that characteristics of the reader, notably working memory capacity and cognitive style, mediate the influence of structure [37].

TABLE VI
RELATING THE FORMALIZATION TO THE EMPIRICAL FINDINGS

Principle/Finding	Way it is accounted for in the formalization
Subjectiveness	By changing the set of patterns, the comparison of tours can be adapted to different preferences or cognitive styles of reviewers.
Principle 1 (group related parts)	A grouping pattern captures the notion of "all related change parts", and the definition of the match set and its "is better than" relation ensure that in a better tour more related change parts are close together.
Principle 2 (provide information before needed)	It is hard to formalize the notion of provided information; not least because information structures in software are often cyclic, e.g., with the caller of a method providing information on why the callee exists and how it is used and the callee providing information about its pre- and postconditions. Currently, reviewers often resort to heuristics like going bottom-up or top-down along the call flow, and these heuristics can be included in the formalization in the grouping pattern's rating function.
Principle 3 (prefer grouping)	A pattern match is only included in the match set if the matched parts are close together, and the rating function is only relevant for matches included in the match set. This is a very strict interpretation of the principle; it was chosen because the participants of the survey rated tours with intervening unrelated change parts low, independent of the distance: For 85 of 113 (75%) respondents a tour with one unrelated change part in between was rated as "not very useful" or "not at all useful", and almost the same number said this for two unrelated change parts in between (82 of 113).
Principle 4 (chunking)	The notion of chunking is formalized by the recursive evaluation of mS on shrunk tours and graphs.
Principle 5 (closest is neighbouring)	This principle is more relevant to the presentation of the change parts in the review tool and therefore not explicitly integrated into the formalization.
Principle 6 (understandable rules)	The grouping patterns as a central part of the formalization can be easily explained to software developers. Furthermore, they can be made explicit to the reviewer.
Macro structure	We noticed that applying the ordering principles generally leads to a sensible macro structure, too, mostly due to the inclusion of the chunking principle. Therefore, we did not take further measures regarding the macro structure.
Importance order	The importance of a change part for review has not been included in the formalization. Instead, we propose to remove clearly unimportant change parts from the review scope and to optimize the order of the remaining for understandability.
Open questions	There are a number of areas where data is lacking for a grounded formalization, e.g., how to best include differing strengths of gradual relations or differing priorities of grouping patterns. Therefore, we took a conservative approach and defined the relation $\geq_T$ to be partial, with the downside that many tours end up as incomparable.

The notion of 'guided tours' has also been proposed for hypertext [38]. In addition, Hammond and Allison [39] suggest the metaphors of "go-it-alone" (similar to the targeted navigation briefly mentioned in Section II-A) and of "map navigation" (similar to the need of our participants to get an overview). An approach to automatically create such guided tours has been proposed by Guinan and Smeaton [40]. Like us, they use patterns to determine the order of the nodes.

C. Empirical Findings on Real-World Code Structure

We expect that developers in long-living projects try to structure their code in a way that helps understanding. Therefore, we looked for empirical results on the order of methods and fields in software systems. We found two: Biegel et al. observed that in many cases, the code adheres to the structure specified in the Java Code Conventions published by Sun/Oracle, and that a clustering by visibility is also quite common [41]. They could also observe semantic clustering (by common key terms), whereas alphabetic order was rare. Geffen and Maoz studied a number of different criteria, also on open-source Java projects but with a stronger focus on call-flow relationships [42]. They found that a "calling" criterion of having a callee after the caller (i.e., top-down) is often satisfied. Regarding the conflicting results on top-down vs bottom-up between the interviews and survey, this can be regarded as a point in favor of top-down. The article of Geffen and Maoz also contains results on a second study: They tested experimentally whether clustering or sorting by call-flow helps to understand code faster. Their results are not statistically significant, but they show a tendency that a random order is worst and a combination of clustering and sorting is best, especially for inexperienced developers.

D. Clustering of Program Fragments and Change Parts

After the importance of grouping in an optimal tour became clear, we started to look at existing approaches for clustering in software. Many clustering approaches exploit structural [43] and similarity relations [44], possibly augmented with latent semantic analysis [45]. Often clustering is performed using randomized meta-heuristics, an approach we regard as incompatible with Principle 6 (understandable rules). In contrast, the ACDC approach of Tzerpos and Holt [46] is based on recognizing patterns in subsystem structures and encouraged us to pursue a similar approach.

Our work is different from most existing approaches in that we are dealing with relations between change parts instead of program fragments. A line of research that also deals with the clustering of change parts is "change untangling" [47], [48]. A number of approaches have been proposed that cluster changes based on heuristics and pattern matching [9], [49], [50], [51]. Future work should check whether these approaches can be adjusted to help in finding an optimal tour.

E. Program Comprehension: Empirical Findings and Theories

A number of theories on the cognitive processes of developers during code comprehension have been proposed. Developers sometimes use 'bottom-up comprehension', i.e., they combine and integrate parts of the program into increasingly complete mental models. On other occasions, they employ 'top-down comprehension', either inference-based by using beacons in the code or expectation-based guided by hypotheses [52]. They switch between these modes depending on their knowledge, the needs of the task, and other factors [53], [54]. The survey by Storey [55] provides further information.

Recent studies looked at the navigation behavior of developers during debugging and maintenance. It was found that 'information foraging theory' provides a more accurate pattern of navigation behavior than hypothesis-driven exploration [56], [57]. In information foraging, developers follow links between program fragments. These links are largely based on dependencies. The importance of links/relations for developer navigation has also been noted in other studies [58], [59], [60]. A comparison of developers with differing experience showed that effective developers navigate by following structural information [61] and make more use of chunking [62].

Storey et al. [63] combined empirical findings from the literature to derive guidelines for tools that support program comprehension. The theory described in the current article paves the way towards such a tool, therefore their guidelines should be partly reflected in our findings. Our approach is mainly meant to enhance bottom-up comprehension, and we regard their elements "reduce the effect of delocalized plans" (by grouping) and "provide abstraction mechanisms" (by hierarchical chunking) to be applicable to our approach. By allowing the reviewers to also explore the source code on their own, some more of their elements can be satisfied.

The cognitive processes during review have been studied by Hungerford et al. for reviewing design documents [64]. They found that a 'Concurrent Across Diagrams' strategy, i.e., reading with frequent switches between related parts of diagrams, seems to be most effective.

F. Reading Techniques

Reading techniques [65], i.e., instructions on how to read a software work product, have been studied extensively in the context of Inspections. The code reading techniques that are most closely related to our work are "abstraction based reading" [66], "use case/usage based reading" [67] and "functionality based reading" [68]. "Abstraction based reading" has been proposed by Dunsmore et al. to overcome problems with delocalization in the review of object-oriented code. It is based on forming summaries of a program in a bottom-up style. "Usage-based reading" and "functionality-based reading" work instead by tracing use cases/functionality in a top-down style. These reading techniques showed promising results in controlled experiments, but some of these could not be consistently replicated in further studies [69], [70], [71].

A commonality that all these reading techniques share with our technique to find optimal tours is that they include structural information, namely call-flow and data-flow. But there are two main differences. The first is technical: We are focusing on change-based code review and therefore on ordering change parts, while the presented reading techniques have been defined based on a single version of the code. The second difference concerns the underlying assumptions: Many reading techniques try to actively guide the reviewer and enforce strict guidelines for the reading process. The evidence whether active guidance is indeed beneficial is inconclusive [72], [73]. Enforcing strict guidelines is regarded as conflicting with theories of software cognition [74] and seen as too rigid by software developers [75]. In our work we take a different position: The reviewer and the review tool form a joint cognitive system [23]. The tool is there to help the reviewers by reducing their cognitive load, but the reviewers are flexible in whether they want to follow the tool's guiding or explore the code change on their own instead.

VII. Future Work

The most important next step is to increase confidence in the theory, by systematically testing it with both controlled experiments and in a real-world setting after inclusion in a code review tool. The possibility that providing an 'optimal' tour could passivate the reviewer and therefore lower review effectiveness is worth investigating.

Open questions to improve the theory are whether and which relation types are (more) important. Further improvements could be obtained by gaining more data on differences between deletions, changes, and additions; especially, we studied deletions only marginally.

The notion of a tour as a permutation of the change parts could also be reconsidered: It could be beneficial to re-visit change parts or even to visit unchanged parts of the code. And re-framing the goal from determining an optimal tour at the start of the review to recommending a good next change part at any time during the review, which allows taking the reviewer's navigation history into account, appears to be worthwhile.

VIII. Conclusion

We studied what makes one change part order better than another for review. We used a mixed methods approach, combining insights from task-guided interviews, an online survey, logs of code reviews in industry, and the literature.

We derived 6 principles that describe what a good order is and how it should be presented: In short, related change parts should be close together, they should be ordered (if the grouping allows it) so that information is provided before it is needed, and the resulting hierarchical clustering should make sense to a human reviewer. We formalized these principles by giving a middle-range theory that defines a partial order relation $\geq_T$ among review tours. It uses the notion of finding patterns in the graph of relations among the change parts. Our propositions are based on the belief that the code reviewer and the review tool form a joint cognitive system so that review efficiency can be increased by moving cognitive load from the reviewer to the tool.

Acknowledgment

The authors would like to thank all participants of the interviews, the pre-tests, and the survey. We furthermore thank Leif Singer for his help with contacting the GitHub users for the survey. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[2] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "A faceted classification scheme for change-based industrial code review processes," in *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*. Vienna, Austria: IEEE, 2016.

[3] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg, Russia: ACM, 2013, pp. 202–212.

[4] M. Bernhart and T. Grechenig, "On the understanding of programs with continuous code reviews," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. San Francisco, CA, USA: IEEE, 2013, pp. 192–198.

[5] T. Baum, H. Leßmann, and K. Schneider, "The choice of code review process: A survey on the state of the practice," in *Product-Focused Software Process Improvement: 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 01, 2017, Proceedings*. Springer, 2017, to appear.

[6] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad, India: ACM, 2014, pp. 345–355.

[7] Github. [Online]. Available: https://github.com

[8] T. Baum and K. Schneider, "On the need for a new generation of code review tools," in *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17*. Springer, 2016, pp. 301–308.

[9] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *Proceedings of the 2015 International Conference on Software Engineering. IEEE Press*, 2015.

[10] D. I. Sjøberg, T. Dybå, B. C. Anda, and J. E. Hannay, "Building theories in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 312–336.

[11] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, 1967.

[12] B. G. Glaser, *Theoretical Sensitivity – Advances in the Methodology of Grounded Theory*. The Sociology Press, 1978.

[13] Q. Zhang, "Improving software development process and project management with software project telemetry," Ph.D. dissertation, University of Hawaii, 2006.

[14] T. Baum, K. Schneider, and A. Bacchelli. (2017) Online material for "On the optimal order of reading source code changes for review". [Online]. Available: http://dx.doi.org/10.6084/m9.figshare.5236150

[15] B. Hanington and B. Martin, *Universal methods of design: 100 ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport Publishers, 2012.

[16] D. Spencer, "Card sorting: a definitive guide," http://boxesandarrows.com/card-sorting-a-definitive-guide/, 2004.

[17] R. Jacob, A. Heinz, and J. P. Décieux, *Umfrage: Einführung in die Methoden der Umfrageforschung*. Walter de Gruyter, 2013.

[18] A. Tversky and D. Kahneman, "Judgment under uncertainty: Heuristics and biases," in *Utility, probability, and human decision making*. Springer, 1975, pp. 141–162.

[19] C. Oezbek and L. Prechelt, "Jtourbus: Simplifying program understanding by documentation that provides tours through the source code," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 64–73.

[20] K. Schneider, "Prototypes as assets, not toys: why and how to extract knowledge from prototypes," in *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society, 1996, pp. 522–531.

[21] S. Biffl, "Analysis of the impact of reading technique and inspector capability on individual inspection performance," in *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*. IEEE, 2000, pp. 136–145.

[22] *Systems and software engineering—Vocabulary ISO/IEC/IEEE 24765: 2010*, IEEE Standards Association and others Std. 24 765, 2010.

[23] A. Walenstein, "Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering," in *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 2003, pp. 185–194.

[24] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 378–389.

[25] B. Floyd, T. Santander, and W. Weimer, "Decoding the representation of code in the brain: An fmri study of code review and expertise," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, 2017.

[26] S. Göpferich, "Comprehensibility assessment using the Karlsruhe comprehensibility concept," *The Journal of Specialised Translation*, vol. 11, no. 2009, pp. 31–52, 2009.

[27] I. Langer, F. S. von Thun, and R. Tausch, *Sich verständlich ausdrücken*, 9th ed. E. Reinhardt, 2011.

[28] L. T. Frase, "Paragraph organization of written materials: The influence of conceptual clustering upon the level and organization of recall." *Learning and instructional Processes*, 1969.

[29] B. J. Meyer, "Reading research and the composition teacher: The importance of plans," *College composition and communication*, pp. 37–49, 1982.

[30] A. C. Graesser, D. S. McNamara, and M. M. Louwerse, "What do readers need to learn in order to process coherence relations in narrative and expository text," *Rethinking reading comprehension*, pp. 82–98, 2003.

[31] W. Kintsch, T. S. Mandel, and E. Kozminsky, "Summarizing scrambled stories," *Memory & Cognition*, vol. 5, no. 5, pp. 547–552, 1977.

[32] R. A. Yaros, "Is it the medium or the message? structuring complex news to enhance engagement and situational understanding by nonexperts," *Communication Research*, vol. 33, no. 4, pp. 285–309, 2006.

[33] J. B. Black and G. H. Bower, "Story understanding as problem-solving," *Poetics*, vol. 9, no. 1-3, pp. 223–250, 1980.

[34] D. S. McNamara, E. Kintsch, N. B. Songer, and W. Kintsch, "Are good texts always better? interactions of text coherence, background knowledge, and levels of understanding in learning from text," *Cognition and instruction*, vol. 14, no. 1, pp. 1–43, 1996.

[35] R. A. Yaros, "Effects of text and hypertext structures on user interest and understanding of science and technology," *Science Communication*, vol. 33, no. 3, pp. 275–308, 2011.

[36] H. Potelle and J.-F. Rouet, "Effects of content representation and readers' prior knowledge on the comprehension of hypertext," *International Journal of Human-Computer Studies*, vol. 58, no. 3, pp. 327–345, 2003.

[37] D. DeStefano and J.-A. LeFevre, "Cognitive load in hypertext reading: A review," *Computers in human behavior*, vol. 23, no. 3, pp. 1616–1641, 2007.

[38] R. H. Trigg, "Guided tours and tabletops: tools for communicating in a hypertext environment," *ACM Transactions on Information Systems (TOIS)*, vol. 6, no. 4, pp. 398–414, 1988.

[39] N. Hammond and L. Allinson, "Travel around a learning support environment: rambling, orienteering or touring?" in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1988, pp. 269–273.

[40] C. Guinan and A. F. Smeaton, "Information retrieval from hypertext using dynamically planned guided tours," in *Proceedings of the ACM conference on Hypertext*. ACM, 1992, pp. 122–130.

[41] B. Biegel, F. Beck, W. Hornig, and S. Diehl, "The order of things: How developers sort fields and methods," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 88–97.

[42] Y. Geffen and S. Maoz, "On method ordering," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, 2016.

[43] S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code." in *IWPC*, vol. 98, 1998, pp. 45–52.

[44] A. Kuhn, S. Ducasse, and T. Girba, "Enriching reverse engineering with semantic clustering," in *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, 2005, pp. 10–pp.

[45] J. I. Maletic and A. Marcus, "Using latent semantic analysis to identify similarities in source code to support program understanding," in *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*. IEEE, 2000, pp. 46–53.

[46] V. Tzerpos and R. C. Holt, "Accdc: an algorithm for comprehension-driven clustering," in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*. IEEE, 2000, pp. 258–267.

[47] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 121–130.

[48] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *Software Analysis, Evolution and Reengineering, 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 341–350.

[49] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 2015, pp. 180–190.

[50] S. Platz, M. Taeumel, B. Steinert, R. Hirschfeld, and H. Masuhara, "Unravel programming sessions with thresher: Identifying coherent and complete sets of fine-granular source code changes," in *Proceedings of the 32nd JSSST Annual Conference*, 2016.

[51] J. Matsuda, S. Hayashi, and M. Saeki, "Hierarchical categorization of edit operations for separately committing large refactoring results," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*. ACM, 2015, pp. 19–27.

[52] M. P. O'Brien and J. Buckley, "Inference-based and expectation-based processing in program comprehension," in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*. IEEE, 2001, pp. 71–78.

[53] A. M. Vans, A. von Mayrhauser, and G. Somlo, "Program understanding behavior during corrective maintenance of large-scale software," *International Journal of Human-Computer Studies*, vol. 51, pp. 31–70, 1999.

[54] A. von Mayrhauser and A. M. Vans, "Industrial experience with an integrated code comprehension model," *Software Engineering Journal*, vol. 10, no. 5, pp. 171–182, 1995.

[55] M.-A. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[56] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008, pp. 1323–1332.

[57] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[58] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 7–18.

[59] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012.

[60] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *Software Engineering, IEEE Transactions on*, vol. 32, no. 12, pp. 971–987, 2006.

[61] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: An exploratory study," *Software Engineering, IEEE Transactions on*, vol. 30, no. 12, pp. 889–903, 2004.

[62] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 361–370.

[63] M.-A. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[64] B. C. Hungerford, A. R. Hevner, and R. W. Collins, "Reviewing software diagrams: A cognitive study," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 82–96, 2004.

[65] V. Basili, G. Caldiera, F. Lanubile, and F. Shull, "Studies on reading techniques," in *Proc. of the Twenty-First Annual Software Engineering Workshop*, vol. 96, 1996, p. 002.

[66] A. Dunsmore, M. Roper, and M. Wood, "Systematic object-oriented inspection – an empirical study," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 135–144.

[67] T. Thelin, P. Runeson, and B. Regnell, "Usage-based reading—an experiment to guide reviewers with use cases," *Information and Software Technology*, vol. 43, no. 15, pp. 925–938, 2001.

[68] Z. Abdelnabi, G. Cantone, M. Ciolkowski, and D. Rombach, "Comparing code reading techniques applied to object-oriented software frameworks with regard to effectiveness and defect detection rate," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 2004, pp. 239–248.

[69] A. Dunsmore, M. Roper, and M. Wood, "The development and evaluation of three diverse techniques for object-oriented code inspection," *Software Engineering, IEEE Transactions on*, vol. 29, no. 8, pp. 677–686, 2003.

[70] M. Skoglund and V. Kjellgren, "An experimental comparison of the effectiveness and usefulness of inspection techniques for object-oriented programs," in *8th International Conference on Empirical Assessment in Software Engineering (EASE 2004)*. IET, 2004.

[71] D. A. McMeekin, "A software inspection methodology for cognitive improvement in software engineering," Ph.D. dissertation, Curtin University of Technology, 2010.

[72] C. Denger, M. Ciolkowski, and F. Lanubile, "Does active guidance improve software inspections? a preliminary empirical study." in *IASTED Conf. on Software Engineering*, 2004, pp. 408–413.

[73] F. Lanubile, T. Mallardo, F. Calefato, C. Denger, and M. Ciolkowski, "Assessing the impact of active guidance for defect detection: a replicated experiment," in *Software Metrics, 2004. Proceedings. 10th International Symposium on*. IEEE, 2004, pp. 269–278.

[74] D. J. Cooper, B. R. von Konsky, M. C. Robey, and D. A. McMeekin, "Obstacles to comprehension in usage based reading," in *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*. IEEE, 2007, pp. 233–244.

[75] D. A. McMeekin, B. R. von Konsky, E. Chang, and D. J. Cooper, "Evaluating software inspection cognition levels using bloom's taxonomy," in *Software Engineering Education and Training, 2009. CSEET'09. 22nd Conference on*. IEEE, 2009, pp. 232–239.